SUBMITTED TO: EIGHTH INTERNATIONAL CONFERENCE ON DATA ENGINEERING

Database Recovery Using Redundant Disk Arrays*

Antoine N. Mourad†
W. Kent Fuchs
Daniel G. Saab

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
1101 W. Springfield Ave
University of Illinois
Urbana, Illinois 61801.

June 6, 1991

Abstract

Redundant disk arrays provide a way for achieving rapid recovery from media failures with a relatively low storage cost for large scale database systems requiring high availability. In this paper we propose a method for using redundant disk arrays to support rapid recovery from system crashes and transaction aborts in addition to their role in providing media failure recovery. A twin page scheme is used to store the parity information in the array so that the time for transaction commit processing is not degraded. Using an analytical model, we show that the proposed method achieves a significant increase in the throughput of database systems using redundant disk arrays by reducing the number of recovery operations needed to maintain the consistency of the database.

*This research was supported in part by the National Aeronautics and Space Administration (NASA) under Contract NAG 1-613 and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Grant N00014-91-J-1283.

†Ph. (217) 244-7180, Fax (217) 244-5686, email: mourad@crhc.uiuc.edu

1 Introduction

In a database system, rapid recovery may be necessary for restoring the database to a consistent state after a failure. Several types of failures can occur. The most typical are transaction aborts which can be due to program errors, deadlocks, or can be user initiated. When a transaction aborts, the recovery manager has to restore all database pages modified by the transaction to their previous state. The second type of failure is a system crash. In this case system tables maintained in main memory are lost. The recovery mechanism has to UNDO all updates made to the database by transactions that were active when the crash occurred and to REDO modifications performed by complete transactions and not yet reflected in the database at the time of the crash.

Another type of failure is media failure. One common way to deal with this type of failure is by periodically generating archive copies of the database and by logging updates to the database performed by committed transactions between archive copies into a redo log file. When a media failure occurs the database is reconstructed from the last copy and the log file is used to apply all updates performed by transactions that committed after the last copy was generated. In such a case, a media failure causes significant down time and the overhead for recovery is quite high. For large systems, e.g., with over 50 disks, the mean time to failure (MTTF) of the permanent storage subsystem can be less than 25 days[1]. Mirrored disks have been employed to provide rapid media recovery [1]. However, disk mirroring incurs a 100% storage overhead which is prohibitive for many applications. Redundant Disk Array (RDA) organizations [2, 3] provide an alternative for maintaining reliable storage. However, even when disk mirroring or RDAs are used, archiving and redo logging may still be necessary to protect the database against operator errors or system software design errors.

In this paper, we present a technique that exploits the redundancy in disk arrays to support recovery from transaction and system failures in addition to providing fast media recovery. This is achieved by using a twin page scheme for storing the parity information making it possible to keep the old version of the parity along with the new version. The old version of the parity is used

[1] Assuming an MTTF of 30,000 hours for each disk.

to undo updates performed by aborted transactions or by transactions interrupted by a system failure.

In Sections 2 and 3 we briefly review several techniques for transaction recovery in database systems and discuss two RDA organizations. In Section 4, we present our database recovery scheme. The results of our performance analysis are detailed in Section 5. Section 6 presents some conclusions.

2 Recovery Techniques

Recovery algorithms typically use some form of logging or shadowing. In the logging approach [4], before a new version (*after-image*) of a record or page is written to the database, a copy of the old version (*before-image*) is placed into a sequential log file. If a transaction aborts or the system crashes, the log file is analyzed and the state of the database is restored. In the shadowing approach the update of a page is placed into a new physical page on disk [5, 6]. The physical pages containing the old versions are released after all updates of the committing transaction have been written to disk. One problem with the shadowing approach is dynamic mapping since it requires maintaining a very large page table which leads to high I/O overhead during normal processing. Another problem is the disk scrambling effect which decreases the sequentiality of disk accesses.

In describing and in analyzing our method, we will use the following taxonomy of database recovery algorithms introduced by Haerder and Reuter [7]. They classify recovery algorithms with respect to the following four concepts:

Propagation[2] of updates. The propagation strategy can be *ATOMIC* in which case any set of updated pages can be propagated to the database in one atomic action. In the $\neg ATOMIC$ case, propagation of updates can be interrupted by a system crash and database pages are updated-in-place.

Page replacement. Two policies can be used: the *STEAL* policy allows pages modified by

[2]Propagation to the database means that the new version is visible to higher level software. Updates can be written to disk without being propagated (e.g., shadowing).

uncommitted transactions to be propagated to the database before end-of-transaction (EOT); the opposite policy is referred to as $\neg STEAL$. No UNDO recovery is necessary with a $\neg STEAL$ policy.

EOT processing. Two categories exist: the *FORCE* discipline requires all pages modified by a transaction to be propagated before EOT; the opposite discipline is called $\neg FORCE$.

Checkpointing Schemes. Checkpointing is used to propagate updates to the database in order to minimize the number of REDO recovery actions to be performed after a crash. In the Transaction Oriented Checkpointing (TOC) scheme, a checkpoint is generated at the end of each transaction. This is equivalent to using the *FORCE* discipline in EOT-processing. Two other types of checkpoints can be used: Transaction Consistent Checkpoints (TCC) are generated during quiescent periods where no transactions are being processed, Action Consistent Checkpoints (ACC) are less restrictive and require that no update statements are processed during checkpoint generation.

3 Redundant Disk Arrays

3.1 Data Striping

Striped disk arrays have been proposed and implemented for increasing the transfer bandwidth in high performance I/O subsystems [8, 9, 10]. In order to allow the use of a large number of disks in such arrays without compromising the reliability of the I/O subsystem, redundancy is sometimes included in the form of parity information [3, 10]. Patterson *et al.* [3] have presented several possible organizations for Redundant Arrays of Inexpensive Disks (RAID). One interesting organization is RAID with rotated parity in which blocks of data are interleaved across N disks while the parity of the N blocks is written on the $N + 1^{st}$ disk. The parity is rotated over the set of disks in order to avoid contention on the parity disk. Figure 1 shows the array organization with four disks. The organization allows both large (full stripe) concurrent accesses or small (individual disk) accesses. In this paper, we concentrate on small read/write accesses. For a small write access, the data block is read from the relevant disk and modified. To compute the new parity, the old parity has to be read, XORed with the new data and XORed with the old data. Then the new data and new parity can be written back to the corresponding disks. Stonebraker *et al.* [11] have advocated the use of

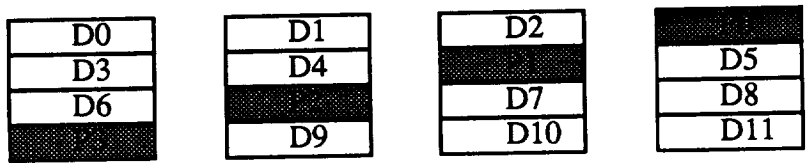

Figure 1: RAID with rotated parity on four disks.

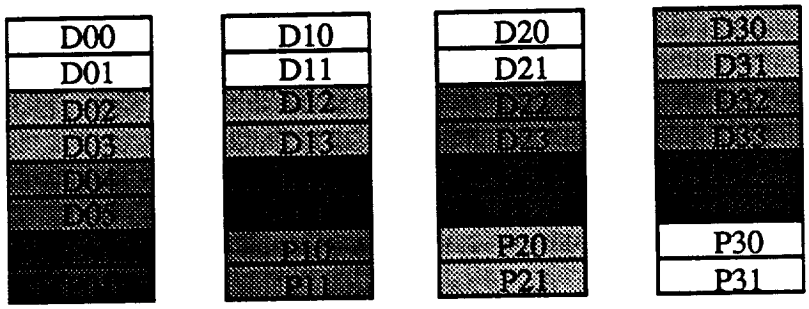

Figure 2: Parity striping of disk arrays.

a RAID organization to provide high availability in database systems.

3.2 Parity Striping

Gray *et al.* [2] studied ways of using an architecture such as RAID in on-line transaction processing (OLTP) systems. They found that because of the nature of I/O requests in OLTP systems, namely a large number of small accesses, it is not convenient to have several disks servicing the same request. Hence, the organization shown in Figure 2 was proposed. It is referred to as parity striping. It consists of reserving an area for parity on each disk and writing data sequentially on each disk without interleaving. For a group of $N+1$ disks, each disk is divided into $N+1$ areas one of these areas on each disk is reserved for parity and the other areas contain data. N data areas from N different disks are grouped together in a *parity group* and their parity is written on the parity area of the $N+1^{st}$ disk.

4 RDA-Based Recovery

In the remainder of this paper, we consider an I/O subsystem that is a collection of redundant disk arrays. The organization of the arrays being either parity striping or data striping (RAID with rotated parity). In the case of data striping we assume that a large striping unit is used in order to

ensure that I/O requests will typically be serviced by a single data disk. We also make the following assumptions: Communication between main memory and the I/O subsystem is performed using fixed size pages; Database pages are updated in place which implies that propagation is $\neg ATOMIC$; A *STEAL* policy is used thus allowing modified pages to be propagated before EOT.

4.1 General Description of the Approach

RDA-based recovery makes use of the parity information present in the disk arrays to undo updates performed by aborted transactions. However, the parity is not sufficient by itself to undo all updates performed by an aborted transaction. Updates that cannot be undone using the parity are dealt with using one of the traditional recovery schemes.

A *page parity group* is the set of pages that share the same parity page. In the following, unless there is ambiguity, we will use the term parity group to denote a page parity group. A parity group can be in one of two states: *clean* or *dirty*. A parity group is dirty when one of its data pages has been modified by a transaction and the modified version has been written back to the database before the transaction modifying it commits (using the notation of Haerder and Reuter, the page has been *stolen* from the buffer). Otherwise the parity group is called clean. Only one modified data page per parity group can be written back to the database by uncommitted transactions without UNDO logging. If additional pages in the parity group have been modified and need to be written back to the database then their before-images must be logged first. A dirty parity group goes back to the clean state when the transaction that caused it to become dirty commits. Figure 3 shows the state transition diagram of a parity group. A table in main memory contains the numbers of all parity groups that are in the dirty state. This table is referred to as the Dirty_Set. It also contains the number of the data page within the group that caused the group to be in the dirty state and the number of the parity page holding the updated parity. Only $\log N$ bits need to be used to store the data page number and one bit for the parity page number. The table is used to check whether a page updated by an active transaction can be written back to disk without UNDO logging.

When a transaction updates a page, that page can be written back to the database without

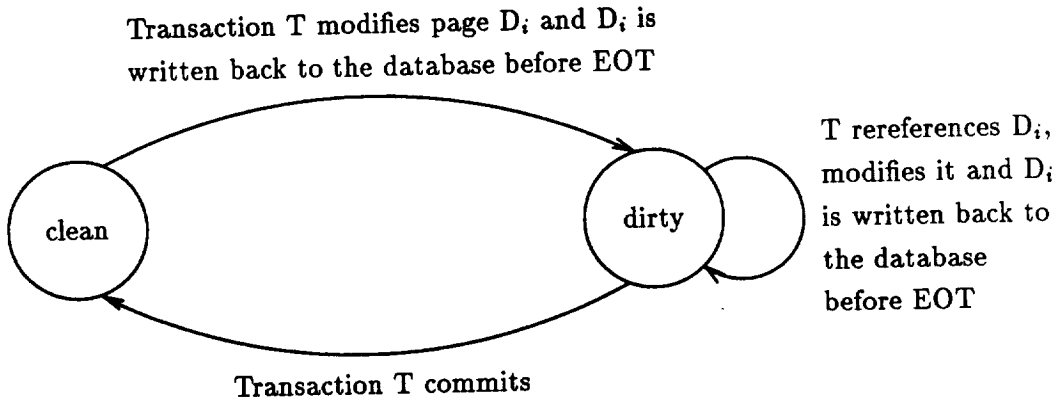

Figure 3: State transition diagram of a page parity group.

UNDO logging if its parity group is clean or if its parity group is dirty and the update is for the same page that caused the group to move into the dirty state, i.e., the same page has been updated, stolen from the buffer then rereferenced by the same transaction, updated and stolen again from the buffer before EOT[3]. Note that this does not affect the degree of concurrency or interfere with the locking policy used in the system. We do not specify when a transaction can or cannot modify a page. We only specify when a modified page can be written back to disk without UNDO logging.

If a single parity page is used, then when a group becomes dirty the old parity information has to be kept in the parity page to be able to recover in case of a transaction failure. That would mean that when the transaction commits, the new parity has to be recomputed in order to update the parity page. That would require reading all the data pages in the group in order to compute the new parity. To avoid that problem a twin page scheme is used for the parity pages. The basic mechanism of the twin page scheme is as follows: one of the parity pages always contains the valid parity of the group while the other page contains obsolete parity information. When a data page is modified in a parity group, the obsolete parity page (P for example) is updated with the new parity of the array. If the transaction performing the update commits then the modified parity page (P) becomes the valid parity page otherwise the other parity page (P') remains the valid parity page and its contents are used to recover the data page that was modified by the failed transaction. Figures 4 and 5 show the data striping organization and the parity striping organization when the

[3]Normally such an event should not occur often since buffer management algorithms are not supposed to replace a page that will be referenced again in the near future.

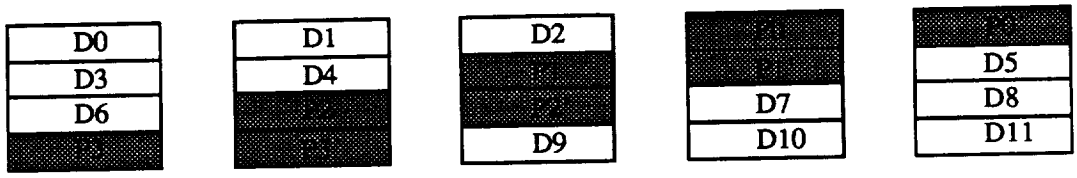

Figure 4: Data striping organization with the twin page scheme for the parity.

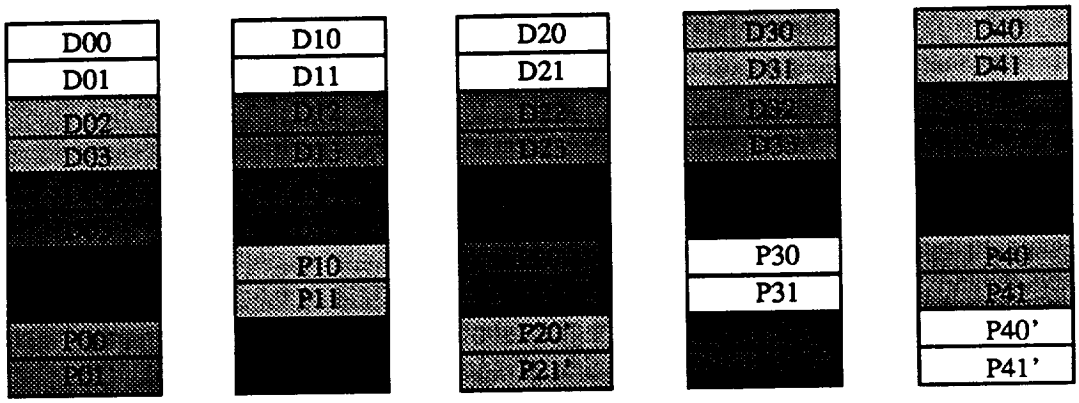

Figure 5: Parity striping organization with the twin page scheme for the parity.

twin page scheme is used for the parity. Twin parity pages are denoted Px and Px' in the data striping case and Pxy and Pzy', with $z = (x+1) \mathrm{mod}(N+2)$, in the parity striping case. Figure 6 shows the contents of a parity group including the twin parity pages. In order to recover the old version of a data page after a transaction abort it is sufficient to XOR the contents of both parity pages and the new data page: $\mathrm{D_{old}} = (\mathrm{P} \oplus \mathrm{P'}) \oplus \mathrm{D_{new}}$. When a parity group is dirty because one of its data pages D_i has been stolen from the buffer and another page D_j needs to be written to disk, UNDO logging must be performed for D_j[4] then both parity pages P and P' need to be updated since when the group is dirty it is necessary to maintain a current parity page reflecting the actual parity of the data on disk and an "old" parity page that would be used to recover the uncommitted data page D_i in case of a transaction abort. In all cases, when writing a data page to disk the corresponding parity page(s) must be updated first.

[4]The before-image of the page in the case of page logging or of the modified record(s) in the case of record logging must be written to a log file.

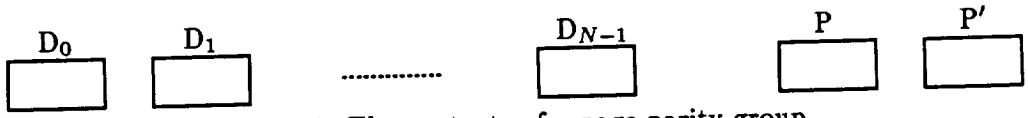

Figure 6: The contents of a page parity group.

4.2 Twin Page Management

The twin parity pages are stored on different disks. This is necessary in order to be able to perform transaction recovery following a disk failure. In order to identify which of the twin parity pages contains the valid parity information, a timestamp is stored in the page header. The page with the highest timestamp contains the valid parity information. When an update is undone after a transaction or system failure, the timestamp of the current parity page is reset to 0. Algorithm **Current_Parity** shown in Figure 7 selects the current parity page. When a data page is updated both parity pages are read and one of them is selected for modification. Then the parity is computed and the modified parity page is written back to disk. In order to avoid reading both parity pages, a bit map can be maintained in main memory indicating which is the current parity page for each of the parity groups in the database. However such a bit map may not survive a system crash. Hence following a crash that destroys the map, algorithm **Current_Parity** will have to be used to identify the current parity page and to reconstruct the bit map. In this case, two bits would have to be used in the bit map for each parity group to code the three possible states: parity page P is the current parity page, parity page P' is the current parity page or the information is not available and algorithm **Current_Parity** has to be used. Following a system crash a background process that runs during idle periods of the system can be initiated to reconstruct the bit map.

Each of the twin parity pages can be in one of four states: *committed, obsolete, working* or *invalid* [12]. A parity page is committed when it contains the last committed parity update. It is obsolete when it contains old committed parity information. It is in the working state when it has been updated by an active transaction, and it is in the invalid state if the last transaction updating it has aborted. Figure 8 shows the state transition diagram of the twin parity pages.

```
Current_Parity(pg)
begin
        Read twin parity pages in parity group pg;
        if Timestamp(P) > Timestamp(P') then
                Current_Parity ← P;
        else
                Current_Parity ← P';
end
```

Figure 7: Algorithm **Current_Parity** determines the current parity page.

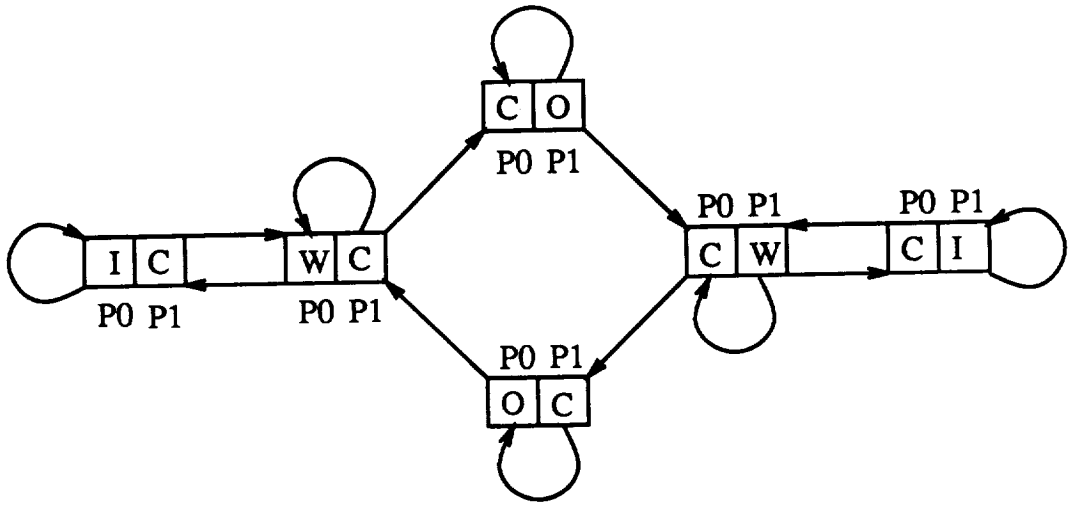

Figure 8: State transition diagram of the twin parity pages.

4.3 Recovery from System Failure

Following a system crash we need to identify which transactions have to be backed out and which pages have been modified *on disk* by those transactions. A Begin-Of-Transaction (BOT) record needs to be written to a log file after the transaction begins and before it writes back any modified pages to disk and an EOT record must be written to the log file when the transaction commits. Modified database pages for which UNDO logging has been performed, can be recovered by reading their before-images from the log. Modified database pages for which UNDO logging has not been performed can be recovered using the parity pages. However information on which pages have been

written to the database without UNDO logging has to be saved in permanent storage. To solve this problem, a technique similar to the one used in TWIST [13] can be employed. In TWIST, a twin page scheme is used to store all database pages, no before-image logging is performed and the same problem of identifying which pages to undo after a crash is encountered. The solution makes use of a log chain which consists of pointers stored in the page headers that link together pages modified by the same active transaction. In our case, only modified pages written back to the database before EOT without UNDO logging will be part of the log chain. The head of the chain though has to be logged along with the transaction id. I/O operations to maintain the log chain can be hidden behind regular I/O requests and do not affect significantly system performance.

5 Performance Analysis

In order to evaluate the benefit of RDA-recovery, we develop an analytical model to evaluate transaction throughput for different algorithms. Since the cost of maintaining parity information in a system with redundant disk arrays is relatively high, we do not advocate the use of RDAs solely for the purpose of supporting transaction and crash recovery. We look at the benefit of using RDA recovery in a system that already needs RDAs for the purpose of rapid media recovery. We do this by comparing the throughput of systems using traditional recovery algorithms and redundant disk arrays to systems with the same recovery algorithms in combination with RDA recovery. We consider both page and record logging and in each case we examine two different recovery algorithms and evaluate the improvement achieved by adding RDA recovery to them. As far as storage is concerned, the extra cost involved in using RDA recovery is that of the twin page scheme for the parity which is $(100/N)\%$ of the initial data storage cost.

RDA recovery reduces the amount of UNDO logging and hence is appropriate for systems using update-in-place which implies $\neg ATOMIC$ propagation and a $STEAL$ policy for page replacement. We therefore restrict ourselves to the analysis of such algorithms. Within this class of algorithms we examine both the $FORCE$ and $\neg FORCE$ strategies for EOT-processing. For algorithms of the type $\neg ATOMIC$, $STEAL$, $FORCE$, only a TOC checkpointing policy makes sense. For algorithms

of the type $\neg ATOMIC$, $STEAL$, $\neg FORCE$, both ACC or TCC checkpoints could be used however algorithms using ACC checkpointing were shown to outperform those using the TCC type[5] [14]. Hence we only look at the former type of checkpointing.

We use the same basic model as the one introduced by Reuter in his evaluation of the performance of several database recovery techniques [14]. We assume that the system is I/O bound and therefore we look only at the number of I/O requests required to perform a given operation. We also assume that the system is running continuously with no periodic shutdown. This implies that all cleanup activities required by the algorithm are accounted for in the cost calculations instead of assuming they are performed by some background process or during shutdown periods.

The workload considered consists of a set of P transactions executing concurrently in the system. Transactions are of two types: *update* or *retrieval*. The fraction of update transactions is f_u. Each transaction accesses s database pages. The fraction of accessed pages that are modified by an update transaction is p_u. To characterize the behavior of the database buffer, we use the communality C which denotes the probability that a page requested by an incoming transaction is present in the buffer. The number of page frames in the buffer is denoted by B. It is assumed that the buffer is sufficiently large so that once a transaction has referenced a page, the page will remain in the buffer until it is no longer needed by the transaction[6].

The cost of recovery after a system crash is denoted by c_s and is measured by the number of page transfers between main memory and the disk subsystem required to perform recovery. The cost of executing a transaction is denoted by c_t. The transaction throughput r_t is defined as the number of transactions processed during an availability interval. An availability interval T is the period between two system crashes. Since all cost measures are evaluated in terms of number of I/O operations, we assume that the availability interval is measured in units of page transfers[7].

If checkpointing is used then the length of a checkpointing interval is denoted by I and is also

[5]Also TCC checkpointing contradicts our assumption of a continuously running system since it requires the establishment of a quiescent point where no update transactions are present in the system.

[6]The page could still be replaced before the transaction commits if a $STEAL$ policy is used, however if it is replaced it will not be rereferenced by the transaction.

[7]Mathematically, T can be defined as follows: $T = \frac{\text{length of availability interval in seconds}}{\text{time to transfer a page to/from disk in seconds}}$

measured in units of page transfers. The cost of generating a checkpoint is denoted by c_c. Assuming that the crash occurs in the middle of a checkpointing interval, the number of page transfers available for processing transactions in an availability interval is $T - c_s - c_c((T - c_s - I/2)/I)$. Hence the throughput is given by:

$$r_t = ((T - c_s)(1 - c_c/I) + c_c/2)/c_t$$

We assume that c_c is independent of I. Hence the optimal checkpointing interval can be easily derived from the following equation [14]:

$$\frac{dr_t}{dI} = (1/c_t)\left(-\frac{dc_s}{dI}(1 - c_c/I) + (T - c_s)(c_c/I^2)\right) = 0. \quad (1)$$

Let c_r denote the cost of updating a retrieval transaction and c_u that of an update transaction. Then c_t can be expressed as follows:

$$c_t = (1 - f_u)c_r + f_u c_u.$$

c_r itself includes two components: the cost of reading pages that are not found in the database buffer and the cost of writing back the replaced pages if they have been modified. Hence:

$$c_r = s(1 - C) + \alpha s(1 - C)p_m, \quad (2)$$

where p_m denotes the probability that the replaced page was modified and α denotes the number of page transfers necessary to perform one write to the disk array. α is equal to 3 or 4 depending on whether or not the old data page is in the buffer at the time of writing the new data. For c_u, we have two additional components which represent the cost of logging the transaction (c_l) and the cost of backing out the transaction (c_b) in the case where an abort occurs. Hence:

$$c_u = s(1 - C) + \alpha s(1 - C)p_m + c_l + p_b c_b, \quad (3)$$

where p_b denotes the probability of an abort.

5.1 Evaluation of the Probability of Logging

We consider a set of K pages that have been modified by active transactions and we compute the expected value of the size of the subset of pages that can be written back to the database without

UNDO logging. N is the number of data pages in a parity group and S is the total number of data pages in the database. We assume that the K pages are randomly chosen from the S pages in the database. Note that by using data striping (RAID) with a large striping unit or parity striping, any sequentiality in database accesses will act in favor of our scheme by distributing the pages accessed over distinct parity groups.

The parity groups in the database are numbered from 1 to S/N. Let X_i, $1 \leq i \leq S/N$, be the random variable whose value is 1 if one of the K pages is a member of parity group i, and 0 otherwise. Let X be the random variable denoting the number of parity groups that contain all K pages. X is also the number of pages that can be directly written back to the database since one page per parity group can be written back. We have:

$$X = \sum_{i=1}^{S/N} X_i.$$

Since the K pages are assumed to be randomly chosen, each parity group has the same probability of being accessed by those K page references. Hence the X_i's are identically distributed. Therefore, the expected value of X is $\mathrm{E}[X] = \sum_{i=1}^{S/N} \mathrm{E}[X_i] = \frac{S}{N}\mathrm{E}[X_1]$. Since X_1 is a Bernoulli random variable, $\mathrm{E}[X_1] = \Pr(X_1 = 1)$ and $\mathrm{E}[X] = \frac{S}{N}(1-\Pr(X_1 = 0))$, which can be written: $\mathrm{E}[X] = \frac{S}{N}\left(1 - \frac{\binom{S-N}{K}}{\binom{S}{K}}\right)$. Hence if K modified, "uncommitted" pages are to be written to the database, the probability of having to log one of those pages is given by:

$$p_l = 1 - \mathrm{E}[X]/K = 1 - \frac{S}{KN}\left(1 - \frac{\binom{S-N}{K}}{\binom{S}{K}}\right). \tag{4}$$

5.2 Page Logging

5.2.1 Algorithm of the Type ¬*ATOMIC*, *STEAL*, *FORCE*, *TOC*

With the *FORCE* discipline, the checkpoint is taken at the end of each transaction. The cost of checkpointing is therefore accounted for in the cost of logging. In the model, we set $c_c = 0$. Given our assumption that pages are not rereferenced by the calling transaction after they have been replaced in the buffer, the cost of writing and logging a page will be the same whether the page is

stolen from the buffer before transaction commit or whether it stays in the buffer until EOT and is then logged and written to the database. Hence we will account for all the costs involved in logging the pages and writing them back to the database as part of the cost of logging. This allows us to set $p_m = 0$ in the expressions for c_r and c_u. The expression for c_l is:

$$c_l = 3 \times sp_u + 4 \times (2sp_u) + 4 \times 4$$

The first term is the cost of writing the pages back to the database. Each write to the disk array costs three I/O operations since, with the FORCE discipline, the old data is kept in the buffer until EOT for the purpose of UNDO logging. The second term is the cost of writing to the UNDO and REDO log files. REDO information is needed only in the case where an operator error or a system software error damages more than one disk in the disk array. The log files are stored separately which makes reading the log to backout aborted transactions less costly. The last term in the expression of c_l is the cost of writing BOT and EOT records to each of the log files.

The probability of having to log a page with RDA recovery is dependent on the number K of pages written back to the database by incomplete transactions. We assume that when a transaction writes back a page to the database before committing, the other concurrent transactions are halfway through writing their own modified pages. Therefore K is equal to half the total number of pages modified by concurrent update transactions. Hence the probability of logging is given by Equation 4 in which K is replaced with[8] $Psf_up_u/2$. With RDA recovery, the formula for the cost of logging becomes:

$$c_l' = (3 + 2p_l)sp_u + 4(sp_u + sp_up_l + 4) + 4(p_l - p_l^{sp_u})$$

The major difference with c_l is that UNDO logging has to be performed only when the parity group is dirty, i.e., with probability p_l. The term $2p_l$ is added to 3 to accounts for the fact that when writing to a dirty parity group both parity pages need to be updated[9]. The last term in the expression of c_l' denotes the cost of writing the log chain header to the log. The header is normally written along with the BOT record in the same page except when the first page written by the

[8] Page logging implies the use of page locking and hence the sets of pages modified by concurrent update transactions are disjoint.

[9] We assume that log file pages and data pages are not mixed in the same parity groups.

transaction to the database has to be logged and not all pages updated by the transaction have to be logged.

To evaluate c_b we assume that a transaction aborts in the middle of processing its pages and that the other concurrent update transactions have also logged half their modified pages. The UNDO log has to be read up to the BOT record of the aborting transaction.

$$c_b = (p_u s/2)(Pf_u) + Pf_u + 4(p_u s/2) + 4$$

The first term is the number of before-images that have to be read from the log. The second term is the number of BOT/EOT records to be read. The third term is the number of page transfers to and from the database to undo the modifications performed by the aborting transaction and the last term account for the writing of a rollback record. With RDA recovery the above formula becomes:

$$c_b' = (p_u p_l s/2)Pf_u + (p_l - p_l^{sp_u})Pf_u + Pf_u + (p_u s/2)(6p_l + 5(1 - p_l)) + 4$$

In the first term the number of logged before-images to be read is now multiplied by p_l. The second term is the expected number of log chain headers to be read from the log. The other major difference is in the fourth term. It is due to the fact that, when recovering a page that has been logged, up to six I/O operations might be necessary since its parity group may still be dirty[10]. On the other hand, if the page has been written to the database without being logged, it is necessary to read both parity pages in its parity group and the "new" data page and then overwrite the database page with the old data and modify the state of the parity page from *working* to *invalid* by resetting the timestamp in its header. Hence five I/O operations will be necessary in the latter case.

After a system crash, only UNDO recovery needs to be performed. Hence the formula for c_s contains the cost of reading the UNDO log file up to the BOT record of the oldest transaction alive at the time of the crash and then overwriting the modifications. The work of the oldest transaction

[10]In this instance and in other instances in the evaluation, we use an upper bound for the costs involved in RDA recovery in order to keep things simple. This will lead to a conservative estimate of the benefit of our method.

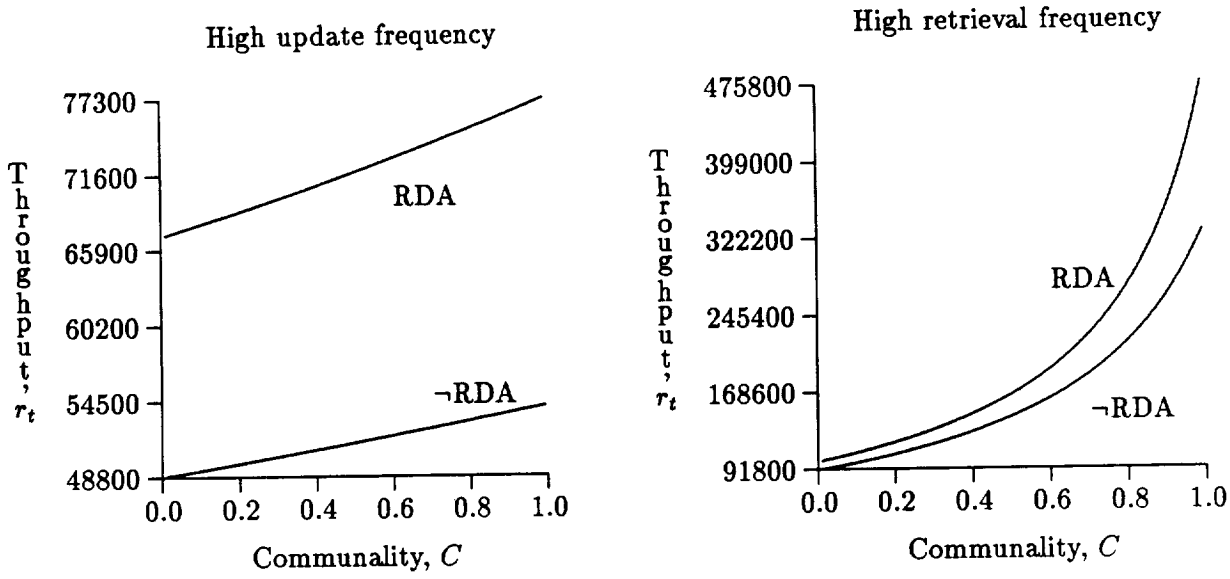

Figure 9: Results for ¬*ATOMIC, STEAL, FORCE, TOC*

alive overlapped with the work of some committed transactions therefore the log records for half the work of about $2Pf_u$ transactions need to be read. Hence the expressions for c_s and c'_s are:

$$c_s = Pf_u(sp_u + 2) + 4(Pf_u p_u s/2)$$

$$c'_s = Pf_u(sp_u p_l + 2(p_l - p_l^{sp_u}) + 2) + Pf_u(p_u s/2)(4p_l + 5(1 - p_l)) + S/N$$

The term S/N is an upper bound for the cost of reconstructing the bit map for the current parity page.

We evaluate the algorithms in two different environments depending on the frequency of update transactions. Figure 9 shows the throughput as a function of the communality C in a system with high update frequency and in a system with high retrieval frequency. As expected the improvement in throughput using RDA recovery is much more significant in the high update frequency environment. For the latter environment and for $C = 0.9$ the increase in throughput is about 42%. All the values for the different parameters of the model, except for N, were taken from [14]. These values are: $B = 300$, $S = 5000$, $N = 10$, $P = 6$, $p_b = 0.01$ and $T = 5.10^6$. For the high update frequency environment, $s = 10$, $f_u = 0.8$ and $p_u = 0.9$ while for the high retrieval frequency environment, $s = 40$, $f_u = 0.1$ and $p_u = 0.3$.

5.2.2 Algorithm of the Type $\neg ATOMIC$, $STEAL$, $\neg FORCE$, ACC

In this case, at EOT, before- and after-images of modified pages are written to the log but the modified pages are not written back to the database. They remain in the buffer until they are replaced. REDO recovery has to be performed after a system crash and ACC checkpointing is used to reduce the amount of REDO during crash recovery.

First we need to evaluate p_m. To do so, we need to compute the number of transactions that successively reference a page during its life in the database buffer. If we look at the stream of references to a page by successive transactions we can see that with probability C the page is referenced when it is in the buffer and with probability $1 - C$ it is referenced when it is not in the buffer. Hence the number of references to the page during its life in the buffer follows a geometric distribution with parameter C which implies that the average number of references to the page while it is in the buffer is $1/(1-C)$. Since the probability of a page being modified by a transaction that references it is $f_u p_u$, the probability of a replaced page being modified during its life in the buffer is[11]:

$$p_m = 1 - (1 - f_u p_u)^{1/(1-C)}$$

The cost of logging is simply the cost of writing before- and after-images of modified pages and the BOT/EOT records to the log:

$$c_l = 4(2sp_u + 2).$$

With RDA recovery, pages that have been stolen from the buffer before EOT do not have to have their before-images logged. Therefore we need to evaluate the probability p_s for a page being stolen. The number of references that could cause a given page to be stolen is $(1 - C)s(P - 1)$ and the probability that any one of those references causes the replacement of the page is $1/(B - Cs)$. Hence the formula for p_s is:

$$p_s = 1 - \left(1 - \frac{1}{B - Cs}\right)^{(1-C)s(P-1)}$$

[11] The same equation for p_m was derived in [14] using a slightly different argument.

In the formula for p_l, the value of K is $Psf_up_up_s/2$. The before-image of a modified page will not be logged with probability $p_s(1-p_l)$. Hence the cost of logging with RDA recovery is:

$$c_l' = 4(sp_u + sp_u(1-p_s(1-p_l)) + 2) + 4(p_l - p_l^{\lceil sp_up_s \rceil}).$$

For the cost of backing out a transaction one difference with the *FORCE* scheme is that the log file contains both before- and after-images which will be read until the BOT record of the aborting transaction is found. Another difference is that with probability C the modified pages to be undone are still in the buffer. Hence:

$$c_b = 2 \times (p_us/2)(Pf_u) + Pf_u + 4p_u(s/2)(1-C) + 4$$

With RDA recovery, the cost of transaction backout becomes:

$$c_b' = 2\times(p_us/2)(Pf_u)+Pf_u+Pf_u(p_l-p_l^{\lceil sp_up_s \rceil})+p_u(s/2)((4+2p_l)(1-C)(1-p_s)+6p_sp_l+5p_s(1-p_l))+4$$

The cost of performing a checkpoint for ¬RDA and for RDA is given by:

$$c_c = 4(Bp_m + 2),$$

$$c_c' = (4 + 2p_l)(Bp_m + 2).$$

To evaluate the cost of recovery after a crash, we assume that a crash occurs in the middle of a checkpoint interval. All transactions executed since the last checkpoint have to be redone. Let r_c denote the number of transactions executed during a checkpoint interval. r_c is given by $r_c = I/c_t$ and the expression for c_s is:

$$c_s = (r_c/2)f_u(c_l/4 + 4sp_u) + Pf_u(c_l/4 + 4(s/2)p_u - 1)$$

The -1 term corresponds to the EOT record which is accounted for in $c_l/4$ but is not read. The cost of recovery from a crash with the RDA recovery technique is:

$$c_s' = (r_c'/2)f_u(c_l'/4 + 4sp_u) + Pf_u(c_l'/4 + (s/2)p_u(4(1-p_s) + 4psp_l + 5p_s(1-p_l)) - 1) + S/N.$$

The value of the optimal checkpointing interval I is obtained by plugging the expression for c_s in Equation 1. This yields:

$$I = (2c_tc_c(T - Pf_u(c_l + 4(s/2)p_u) - Pf_u)/(f_u(c_l + 4sp_u)))^{1/2}.$$

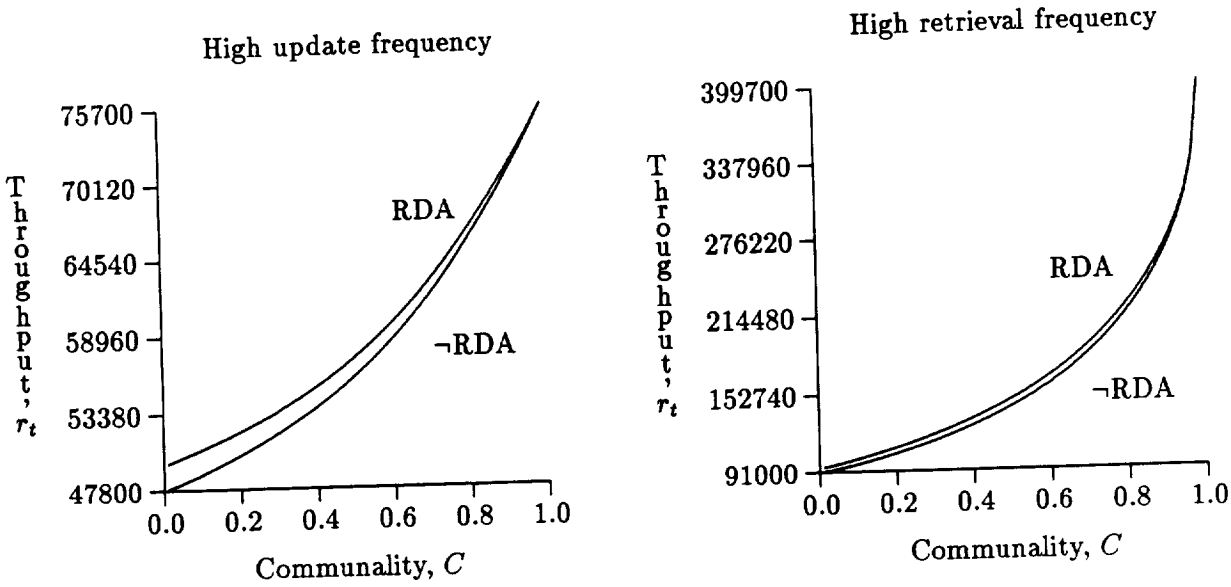

Figure 10: Results for $\neg ATOMIC$, $STEAL$, $\neg FORCE$, ACC

The formula for I in the case of RDA recovery is derived in a similar fashion. The value of α in the expressions of c_r and c_u is 4 for ¬RDA and $4 + 2p_l$ for RDA because with the $\neg FORCE$ discipline, when replacement takes place the old version of the data is not available any more in the buffer.

Figure 10 shows the results for both environments. It can be seen that the improvement is not significant in this case. However the interesting result is that while without RDA recovery, the $\neg FORCE$, ACC type algorithm outperforms the $FORCE$, TOC scheme, when RDA recovery is used, the situation is reversed and the latter algorithm outperforms the former by a significant margin.

5.3 Record Logging

In this section we look at recovery algorithms in which only modified records are logged. The unit of transfer between main memory and secondary storage is still a page however, when logging is performed, logged records are encapsulated into pages and then written to the log file. Some additional parameters of the system need to be introduced for the analysis of record logging: d denotes the number of update statements per transaction; r denotes the average length (in bytes) of a long log entry such as a data record; e denotes the average length of a short log entry such as a table entry; l_{bc} denotes the length of the BOT and EOT records; l_p denotes the length of a

physical page; l_h denotes the length of a log chain header. The values for the first five parameters are taken from [14]. These values are: $d = 3$ for high update frequency environments and $d = 8$ for low update frequency environments, $r = 100$, $e = 10$, $l_{bc} = 16$ and $l_p = 2020$. The value for l_h was set to 4. Assuming that each update statement causes one long log entry and that $s > d$, the average length of a log entry can be derived [14]:

$$L = (dr + (s - d)e)/s.$$

5.3.1 Algorithm of the Type ¬*ATOMIC*, *STEAL*, *FORCE*, *TOC*

With record logging, the locking granule can be less than a page. We assume that record locking is used in order to enhance concurrency. This implies that the total number of pages modified by a given set of P concurrent transaction is not the same as for the above algorithms for which page locking was assumed. We will denote this number by s_u. An expression for s_u is derived in the Appendix. The value of K in the expression of p_l is $s_u/2$. We assume that group commit is used so that log records from different transactions can be grouped in the same page and written to the log. The derivations of the cost equations are similar to those in Section 5.2.1. We simply list the equations without detailed explanation.

$$\begin{aligned}
c_l &= 3sp_u + 4 \times 2(2l_{bc} + sp_u(l_{bc} + L))/l_p \\
c'_l &= (3 + 2p_l)sp_u + 4(2l_{bc} + sp_u(l_{bc} + L))/l_p + 4(2l_{bc} + sp_u(l_{bc} + L)p_l + (l_{bc} + l_h)(p_l - p_l^{sp_u}))/l_p \\
c_b &= Pf_u(l_{bc} + sp_u(l_{bc} + L)/2)/l_p + 4(p_u s/2) + 4 \\
c'_b &= Pf_u(l_{bc} + sp_u(l_{bc} + L)p_l/2 + (l_{bc} + l_h)(p_l - p_l^{sp_u}))/l_p + (p_u s/2)(6p_l + 5(1 - p_l)) + 4 \\
c_s &= Pf_u(2l_{bc} + sp_u(l_{bc} + L))/l_p + 4Pf_u(p_u s/2) \\
c'_s &= Pf_u(2l_{bc} + sp_u(l_{bc} + L)p_l + 2(l_{bc} + l_h)(p_l - p_l^{spu}))/l_p + (Pf_u p_u s/2)(4p_l + 5(1 - p_l))
\end{aligned}$$

Figure 11 shows the throughput for the *FORCE*, *TOC* type of algorithms with and without RDA recovery as a function of the communality in the buffer for the case of record logging.

5.3.2 Algorithm of the Type ¬*ATOMIC*, *STEAL*, ¬*FORCE*, *ACC*

The cost equations for this case can be derived using the results of Sections 5.2.2 and 5.3.1. The value of K in the expression for p_l is $s_u p_s/2$.

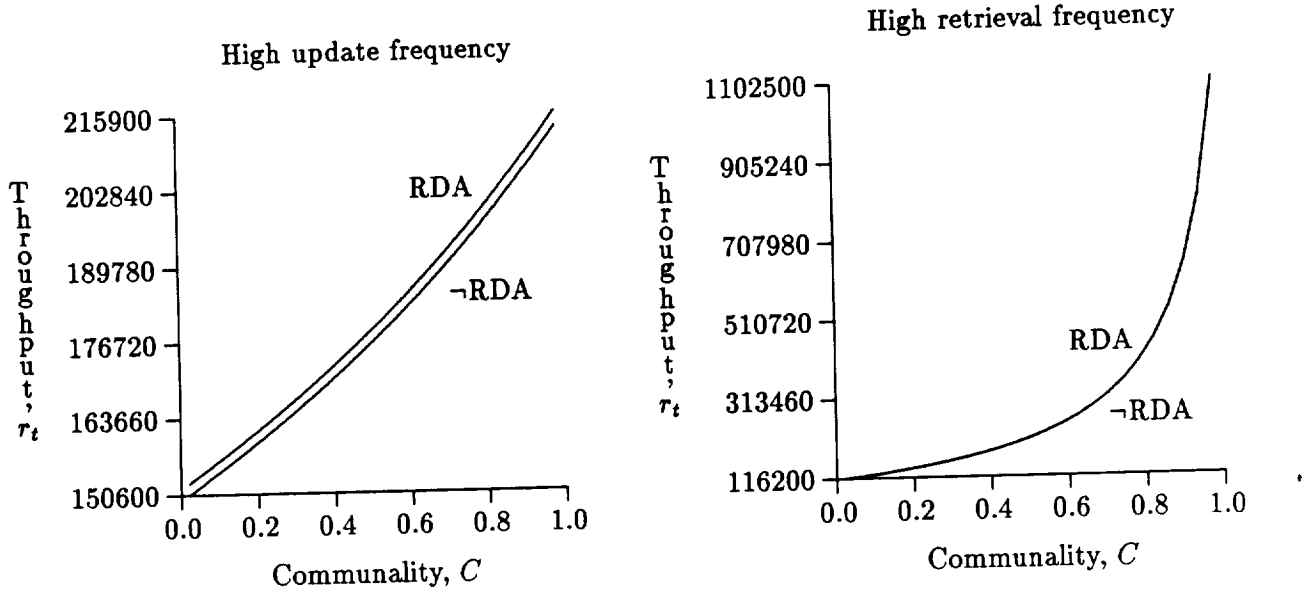

Figure 11: Results for $\neg ATOMIC$, $STEAL$, $FORCE$, TOC, in the case of record logging.

$$
\begin{aligned}
c_l &= 4(2l_{bc} + sp_u(l_{bc} + 2L))/l_p \\
c'_l &= 4(2l_{bc} + sp_u(l_{bc} + L(2 - p_s(1 - p_l))) + (l_{bc} + l_h)(p_l - p_l^{\lceil sp_u p_s \rceil}))/l_p \\
c_b &= Pf_u(c_l/8) + 4p_u(s/2)(1 - C) + 4 \\
c'_b &= Pf_u(c'_l/8) + p_u(s/2)((4 + 2p_l)(1 - C)(1 - p_s) + 6p_s p_l + 5p_s(1 - p_l)) + 4 \\
c_s &= (r_c/2)f_u(c_l/4 + 4sp_u) + Pf_u(c_l/4 + 4p_u(s/2)) \\
c'_s &= (r_c/2)f_u(c'_l/4 + 4sp_u) + Pf_u(c'_l/4 + p_u(s/2)(5p_s(1 - p_l) + 4(1 - p_s(1 - p_l))))
\end{aligned}
$$

The equations for c_c and c'_c are the same as in Section 5.2.2. The equations for c_r and c_u need to be modified to account for the extra cost involved in logging modified records in pages stolen from the buffer before EOT. The modified record of a stolen page needs to be written to the log before the page can be replaced. Let p_i denote the proportion of replaced pages modified by uncommitted transactions. We have $p_i = s^*_u/(B - Cs)$, where s^*_u is the number of pages in the buffer modified by the concurrently executing transactions as seen by an incoming transaction. s^*_u is obtained by replacing P with $P - 1$ in the expression for s_u. This gives the following equations for c_r and c'_r, the equations for c_u and c'_u are obtained in a similar fashion:

$$
\begin{aligned}
c_r &= s(1 - C) + 4s(1 - C)(p_m + 2p_i) \\
c'_r &= s(1 - C) + 4s(1 - C)(p_m + 2p_i p_l)
\end{aligned}
$$

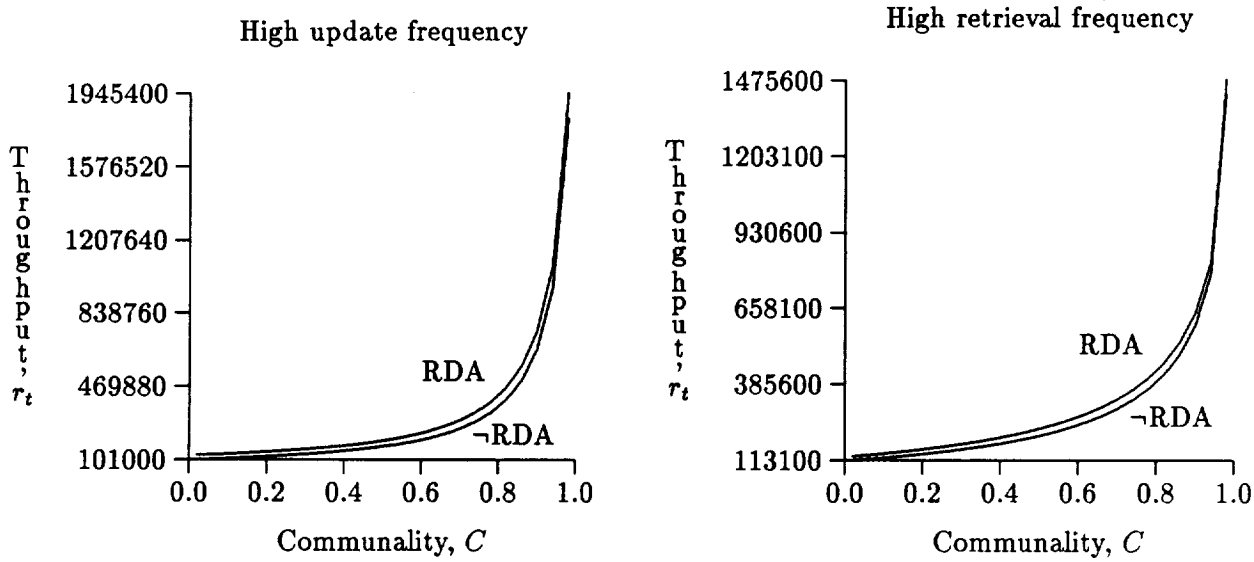

Figure 12: Results for ¬*ATOMIC*, *STEAL*, ¬*FORCE*, *ACC*, in the case of record logging.

Figure 12 shows the throughput for the ¬*FORCE*, *ACC* type of algorithms with and without RDA recovery as a function of the communality in the buffer for both evaluation environments. Unlike the page logging case, ¬*FORCE*, *ACC* scheme performs much better than the *FORCE*, *TOC* scheme for the range of values of C encountered in typical applications [15]. Also, for the ¬*FORCE*, *ACC* algorithm, the increase in throughput achieved by using RDA recovery is higher than for the same algorithm with page logging. This is the case because, with record logging, the cost of logging the updates of a stolen page is high relatively to the cost of logging non stolen pages and RDA recovery reduces that cost by eliminating the need for logging in most cases. For example, for the high update frequency environment and for $C = 0.9$, the increase in throughput is about 14%. The benefit of RDA recovery increases with the amount of work performed by each transaction. Figure 13 shows the percent increase in throughput achieved by RDA recovery as a function of the number of pages accessed by each transaction (s) for the high update frequency environment with $C = 0.9$.

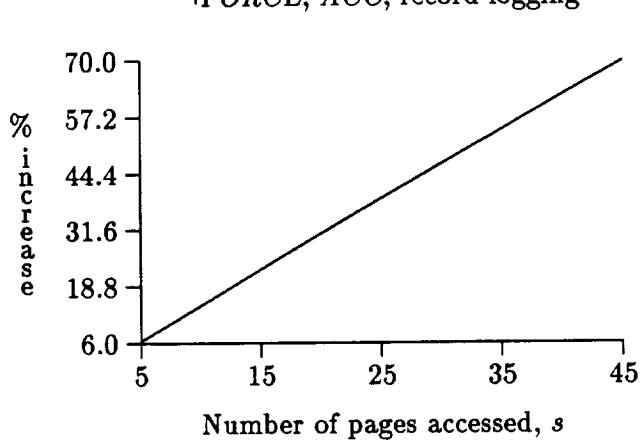

Figure 13: Benefit of RDA recovery as a function of the number of pages referenced by a transaction.

6 Conclusions

In this paper, we have presented a scheme that uses redundant disk arrays to achieve rapid recovery from media failures in database systems and simultaneously provide support for recovery from transaction aborts and system crashes. The redundancy present in the array is exploited to allow a large fraction of pages modified by active transactions to be written to disk and updated in place without the need for undo logging thus reducing the number of recovery actions performed by the recovery component. The method uses a twin page scheme to store the parity information so that it can be efficiently used in transaction undo recovery. The extra storage used is about $(100/N)\%$ of the size of the database, N being the number of disks in the array.

We used a detailed analytical model to evaluate the benefit of our scheme in a system equipped with redundant disk arrays. We found that, in the case of page logging, a *FORCE, TOC* algorithm combined with RDA recovery significantly outperforms a *FORCE, TOC* algorithm without RDA recovery as well as *¬FORCE, ACC* type of algorithms. In the case of record logging, we found that a *¬FORCE, ACC* algorithm performs best and that the addition of RDA recovery to it improves significantly its performance especially for transactions with a large number of updated pages.

Appendix

Derivation of the Formula for s_u

s_u is the number of pages in the buffer updated by a set of P concurrent transactions. Let $S^{(k)}$ denote the number of pages in the buffer updated by k update transactions. Since there are Pf_u update transactions executing concurrently in the system, we have $s_u = S^{(Pf_u)}$. If we number the Pf_u update transaction from 1 to Pf_u in the order of their entry in the system. then when the kth update transaction enters the system, it will find Csp_u of the sp_u pages it needs to modify already in the buffer. We make the assumption that out of those pages, $Csp_u \times S^{(k-1)}/B$ belong to the $k-1$ update transaction already executing in the system[12]. Hence, we have the following recurrence equation:

$$S^{(k)} - S^{(k-1)} = sp_u(1 - CS^{(k-1)}/B)$$

Using $S^{(1)} = sp_u$, we obtain $s_u = S^{(Pf_u)} = \frac{B}{C}(1 - (1 - Csp_u/B)^{Pf_u})$.

References

[1] D. Bitton and J. Gray, "Disk shadowing," in *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 331–338, Sept. 1988.

[2] J. Gray, B. Horst, and M. Walker, "Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput," in *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 148–161, Aug. 1990.

[3] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, pp. 109–116, June 1988.

[4] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system R database manager," *ACM Computing Surveys*, vol. 13, no. 2, pp. 223–242, 1981.

[5] J. Kent and H. Garcia-Molina, "Optimizing shadow recovery algorithms," *IEEE Trans. Software Engineering*, vol. 14, pp. 155–168, Feb. 1988.

[6] R. A. Lorie, "Physical integrity in a large segmented database," *ACM Trans. Database Systems*, vol. 2, pp. 91–104, Mar. 1977.

[7] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, pp. 287–317, Dec. 1983.

[12] Update transactions can share pages because record logging is used instead of page logging.

[8] M. Y. Kim, "Synchronized disk interleaving," *IEEE Trans. Computers*, vol. C-35, pp. 978–988, Nov. 1986.

[9] M. Livny, S. Khoshafian, and H. Boral, "Multi-disk management algorithms," in *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 69–77, May 1987.

[10] K. Salem and H. Garcia-Molina, "Disk striping," in *Proceedings of the IEEE International Conference on Data Engineering*, pp. 336–342, Feb. 1986.

[11] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS," in *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 318–330, Sept. 1988.

[12] K.-L. Wu and W. K. Fuchs, "Rapid transaction-undo recovery using twin-page storage management," in *Proceedings of IEEE Compsac*, pp. 295–300, Nov. 1990.

[13] A. Reuter, "A fast transaction-oriented logging scheme for UNDO recovery," *IEEE Trans. Software Engineering*, vol. SE-6, pp. 348–356, July 1980.

[14] A. Reuter, "Performance analysis of recovery techniques," *ACM Transactions on Database Systems*, vol. 9, pp. 526–559, Dec. 1984.

[15] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems*, vol. 9, pp. 560–595, Dec. 1984.